Using MMX™ Instructions to Implement a Column Filter

Information for Developers and ISVs

From Intel® Developer Services
www.intel.com/IDS

March 1996

CONTENTS

1.0. INTRODUCTION

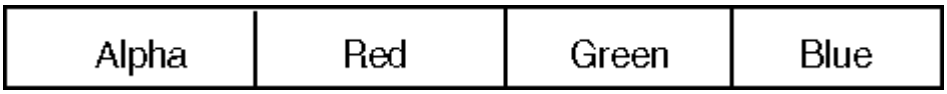
The Intel Architecture (IA) media extensions include single-instruction, multi-data (SIMD) instructions. These instructions are optimized for signal and image processing applications. This application note provides an example of a column filter implemented using MMX instructions.

MMX technology provides the capability to initiate a pipelined multiply instruction every clock cycle (one instruction per clock throughput), with a latency of three clock cycles. This capability, along with the ability to operate on multiple pieces of data in parallel, accounts for significant performance gains. For example, you can perform three complete multiplications of the individual components of three separate 32-bit color values (8-bit RGB and alpha) in six clock cycles. Contrast this to the IMUL instruction (which cannot be pipelined) which can take as many as 10 clock cycles to perform one multiplication of one component.

2.0. THE COLUMN FILTER ALGORITHM

There are several types of column filter algorithms to choose from; your choice depends on the format of the input data. The column filter described in this application note performs multiplications of each 32-bit pixel value in a bitmap by the appropriate filter coefficient, and accumulates seven such multiplications to obtain the final pixel value. Thus, the filter length (or number of filter coefficients) is fixed at seven. Each unsigned 8-bit (RGB and alpha) component of the pixel is multiplied by a signed filter coefficient value. The format of each pixel is shown in Figure 1.

Figure 1. Pixel Format Within the Bitmap

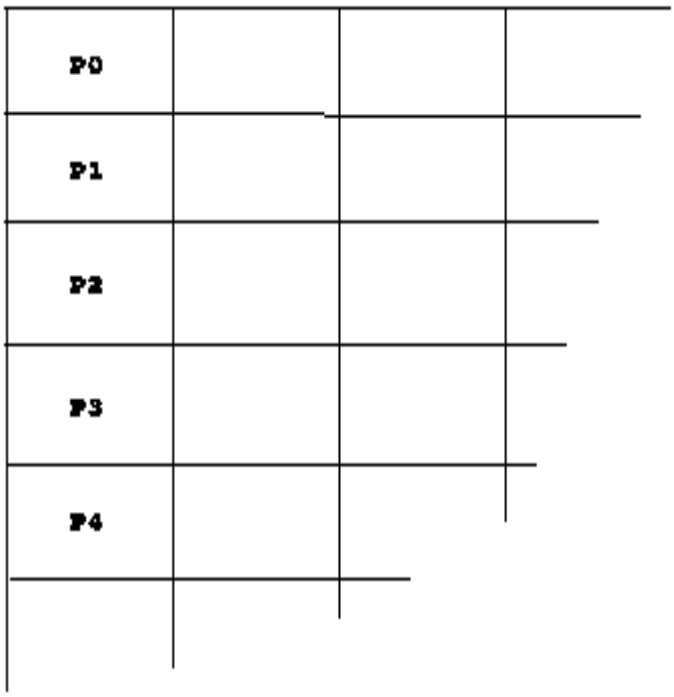


Filter coefficients represent fractional fixed-point values, with the implied binary point located to the left of the most significant bit position. The results of the multiplications of the pixels and filter coefficients are added to the values of similar multiplications of pixels following the first pixel, within the same column.

After the summations are complete, rounding is performed in which binary 0.1 (decimal 0.5) is added to round the result to the nearest whole bit. This final result is written back as the new value of a pixel, representing a weighted average of the pixels involved. This process is then repeated for every pixel in the array. Figure 2 illustrates the C language code equivalent and the calculations necessary to obtain the new values (except for rounding) for pixel 0 (P0) and pixel 1 (P1). Every pixel in the array is calculated in a similar fashion.

Figure 2. Column Filter Algorithm

```
unsigned int SrcBitmap[ROWS][COLS];
unsigned int DstBitmap[ROWS][COLS];
unsigned int Filter_Coeff[FILTER_LENGTH];
unsigned int A, R, G, B;
unsigned int r, c, h;
for (r = 0; r < COLS; r++)   for (c = 0; c < ROWS - FILTER_LENGTH; c++) { A = B = G =
B = 0; for (h = 0; h < FILTER_LENGTH; h++) { A += ((SrcBitmap[r + h][c] >> 24) &
0xFF) * Filter_Coeff[h];
      R += ((SrcBitmap[r + h][c] >> 16) & 0xFF) * Filter_Coeff[h];
      G += ((SrcBitmap[r + h][c] >> 8)  & 0xFF) * Filter_Coeff[h];
      B += ( SrcBitmap[r + h][c]        & 0xFF) * Filter_Coeff[h];
    }
    A += 0x0080;
    R += 0x0080;
    G += 0x0080;
    B += 0x0080;
    DstBitmap[r][c] = ((A << 16) & 0xFF000000) |   ((R << 8) & 0x00FF0000) | (G &
0x0000FF00) | ((B >> 8)  & 0x000000FF);
  }
```



P0 = P0 x H0 + P1 x H1 + P2 x H2 + P3 x H3 + P4 x H4
P1 = P1 x H0 + P2 x H1 + P3 x H2 + P4 x H3 + P5 x H5

3.0. IMPLEMENTING THE COLUMN FILTER

The processor loops through the source array and performs the necessary multiplications and additions on each pixel in order to compute the required values. In order to take advantage of the improvements provided by MMX technology, loop unrolling was used to expose concurrent operations that could be performed on more than one pixel at a time. In this implementation, the loop was unrolled three times, which seemed to offer the best register utilization.

To gain optimum efficiency, the number of memory accesses should be minimized in order to speed up the algorithm. This implementation reads each source pixel value only once and performs all possible operations on each pixel as it is read.

During the development of this implementation, intermediate values were stored in a temporary array, which was the size of the original bitmap. If an intermediate value is stored to memory, it is not allocated in the data cache since the Pentium® Pro processor does not implement a write-allocate cache (a line in the cache is not allocated when data is written). Hence, an external bus cycle is generated when the intermediate result is first stored, and again when the intermediate result is read back to continue the calculation. This creates excessive bus activity. A limited set of temporary variables was then used; in this case nine. This significantly reduced external memory accesses, because after the first read from each of the nine temporary variables, they became allocated in the cache and hit in the cache on every access thereafter.

Pseudocode for this implementation is shown in Example 1.

Example 1. Pseudocode for the Column Filter Implementation

```
#define NUM_COLUMNS 72
c = current column counter
r = current row counter
hy = filter coefficient 'y'
BM[r,c] = pixel in source bitmap at row 'r' and column 'c'
T[x]= temporary variable T[x] used to store intermediate results
For(c = 0; c < NUM_COLUMNS; c++)
precondition:
        T[0] =        BM[0,c] * h0 + BM[1,c] * h1 + BM[2,c] * h2
        T[1] =                       BM[1,c] * h0 + BM[2,c] * h1
        T[2] =                                      BM[2,c] * h0
        T[3] = T[0] + BM[3,c] * h3 + BM[4,c] * h4 + BM[5,c] * h5
        T[4] = T[1] + BM[3,c] * h2 + BM[4,c] * h3 + BM[5,c] * h4
        T[5] = T[2] + BM[3,c] * h1 + BM[4,c] * h2 + BM[5,c] * h3
        r = 6;
        do
        {
inner_loop:
               T[0] = T[3] + BM[r,c] * h6
               T[1] = T[4] + BM[r,c] * h5 + BM[r+1,c] * h6
               T[2] = T[5] + BM[r,c] * h4 + BM[r+1,c] * h5 + BM[r+2,c] * h6
               T[3] = T[6] + BM[r,c] * h3 + BM[r+1,c] * h4 + BM[r+2,c] * h5
               T[4] = T[7] + BM[r,c] * h2 + BM[r+1,c] * h3 + BM[r+2,c] * h4
               T[5] = T[8] + BM[r,c] * h1 + BM[r+1,c] * h2 + BM[r+2,c] * h3
               T[6] =        BM[r,c] * h0 + BM[r+1,c] * h1 + BM[r+2,c] * h2
               T[7] =                       BM[r+1,c] * h0 + BM[r+2,c] * h1
```

```
                T[8] =                                          BM[r+2,c] * h0
                BM[r-6,c] = (T[0] + 0x80) >> 8;
                BM[r-5,c] = (T[1] + 0x80) >> 8;
                BM[r-4,c] = (T[2] + 0x80) >> 8;
                r += 3;
        } while (r < (NUM_ROWS - 1));
}
```

As shown in Example 1, the preconditioning section begins first by reading in the first three pixels of the current column, as signified by *BM[0,c]*, *BM[1,c]*, and *BM[2,c]*. The first three temporary variables (*T[0]*, *T[1]*, and *T[2]*) are initialized with the appropriate multiply-accumulate operations. The next three pixels are read in, as signified *by BM[3,c]*, *BM[4,c]*, and *BM[5,c]*. The second set of three temporary variables (*T[3]*, *T[4]*, and *T[5]*) are initialized with the sum of the previous temporary variables, and the appropriate multiply-accumulate operations. Finally, the last set of three temporary variables *(T[6]*, *T[7]*, and *T[8]*) are initialized based upon the appropriate multiply-accumulate operations.

At this point, all possible (and necessary) calculations involving the first six pixels of the column have been completed. To continue processing, the inner loop must begin with the pixel at row six of the current column. This is why *r* is initialized to six before the inner loop is entered. The inner loop carries out operations similar to those accomplished during preconditioning for the remaining pixels in the given column.

Each iteration of the inner loop causes three additional pixels to be read from the source bitmap, and all possible calculations to be completed with these pixel values. At the end of the inner loop, temporary variables *T[0]*, *T[1]*, and *T[2]* contain values that are ready for rounding, and subsequent writing to the destination bitmap. Rounding is accomplished by adding 80H (decimal 0.5) to each of these temporary values and then shifting right by eight bits. Shifting effectively truncates the final result to a whole number, throwing away all bits to the right of the implicit binary point. The row counter is incremented by three and the inner loop is ready for the next iteration.

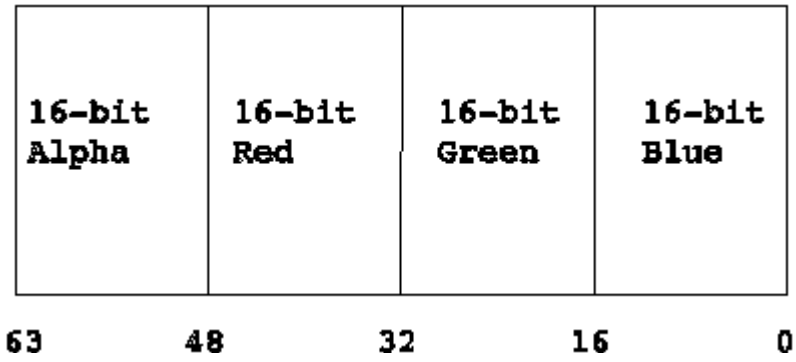
Unrolling the loop three times offers the best register utilization. Unrolling the loop four times was tried, but too many values stored in memory results in an inefficient implementation. It is possible that unrolling the loop by two is more efficient than the implementation described here. Due to time constraints, this option was not explored.

3.1. Unpacking the Data

MMX technology implements eight 64-bit general purpose registers. These registers are aliased to the floating-point registers, so it is not efficient to intermix floating-point operations with MMX technology operations. Each of these registers can operate on data as 8-, 16-, 32-, or 64-bit quantities. For instance, if an instruction operates on 16-bit quantities, it will operate on four 16-bit fields within one MMX register at once (*4 x 16 bits = 64 bits*).

Pixel data is stored in physical memory as shown in Figure 1, with each pixel packing four 8-bit values. In order to perform the necessary operations, each 8-bit component must be zero-extended to 16 bits so it can be subsequently multiplied by the proper filter coefficient. To accomplish unpacking, the PUNPCKLBW instruction was used. This instruction unpacks a pixel value from the format shown in Figure 1 and converts each 8-bit unsigned component of the pixel into 16-bit zero-extended values using up the four 16-bit fields of an MMX register. This is illustrated in Figure 4.

Figure 4. Pixel Format After Unpacking



3.2. Loop Preconditioning

This implementation employs a loop preconditioning step before the inner loop is entered. The preconditioning serves to initialize intermediate results in order to increase the efficiency of the inner loop. For a given column, preconditioning begins by reading the first three pixels of the column from the source bitmap. Assuming a filter with seven filter coefficients, the equation representing the first output pixel in the current column is as follows (except for rounding):

$$T[0]=BM[0,c]*h0+BM[1,c]*h1+BM[2,c]*h2+BM[3,c]*h3+BM[4,c]*h4+BM[5,c]*h5+BM[6,c]*h6$$

Since only three pixels have been read (*BM[0,c]*, *BM[1,c]* and *BM[2,c]*), only three multiplications can be performed before the intermediate result must be stored to a temporary variable *T[0]*. A similar procedure is carried out for pixel *BM[1,c]*, which is represented as the following:

$$T[1]=BM[1,c]*h0+BM[2,c]*h1+BM[3,c]*h2+BM[4,c]*h3+BM[5,c]*h4+BM[6,c]*h5+BM[7,c]*h6$$

Here only two multiplications can be performed because only three pixels have been read, of which only two are important to this calculation (*BM[1,c]* and *BM[2,c]*). A similar procedure is also carried out for the pixel in row two of the current column.

The next three pixels of the column are now read.

This process allows additional multiply-accumulate operations for pixels in rows 0 through two of the current column, as well as for a new set of similar calculations to begin for pixels in rows three through five of the current column.

At the end of the preconditioning step, the following calculations will be completed:

```
T[0] = don't care
T[1] = don't care
T[2] = don't care
T[3] = BM[0,c]*h0+BM[1,c]*h1+BM[2,c]*h2+BM[3,c]*h3+BM[4,c]*h4+BM[5,c]*h5
T[4] = BM[1,c]*h0+BM[2,c]*h1+BM[3,c]*h2+BM[4,c]*h3+BM[5,c]*h4
T[5] = BM[2,c]*h0+BM[3,c]*h1+BM[4,c]*h2+BM[5,c]*h3
T[6] = BM[3,c]*h0+BM[4,c]*h1+BM[5,c]*h2
T[7] = BM[4,c]*h0+BM[5,c]*h1
T[8] = BM[5,c]*h0
```

3.3. The Inner Loop

Each iteration through the inner loop completes processing for three pixels. The inner loop continues from the point where preconditioning left off, and loops through the remaining pixels in the column. Each iteration through the inner loop causes three additional pixels to be read from the source bitmap. At the end of each iteration, the following calculations will be completed:

```
T[0] = BM[r-6,c]*h0+BM[r-5,c]*h1+BM[r-4,c]*h2+BM[r-3,c]*h3+BM[r-2,c]*
       h4+BM[r-1,c]*h5+BM[r,c]*h6
T[1] = BM[r-5,c]*h0+BM[r-4,c]*h1+BM[r-3,c]*h2+BM[r-2,c]*h3+BM[r 1,c]*
       h4+BM[r,c]*h5+BM[r+1,c]*h6
T[2] = BM[r-4,c]*h0+BM[r-3,c]*h1+BM[r-2,c]*h2+BM[r-1,c]*h3+BM[r,c]*
       h4+BM[r+1,c]*h5+BM[r+2,c]*h6
T[3] = BM[r-3,c]*h0+BM[r-2,c]*h1+BM[r-1,c]*h2+BM[r,c]*h3+BM[r+1,c]*h4+BM[r+2,c]*h5
T[4] = BM[r-2,c]*h0+BM[r-1,c]*h1+BM[r,c]*h2+BM[r+1,c]*h3+BM[r+2,c]*h4
T[5] = BM[r-1,c]*h0+BM[r,c]*h1+BM[r+1,c]*h2+BM[r+2,c]*h3
T[6] = BM[r,c]*h0+BM[r+1,c]*h1+BM[r+2,c]*h2
T[7] = BM[r+1,c]*h0+BM[r+2,c]*h1
T[8] = BM[r+2,c]*h0
```

The value *r* (row) shown above is the row count at the beginning of the iteration. At the end of each iteration, the values contained in *T[0]*, *T[1]*, and *T[2*] correspond to the new pixel values for the pixels at row *r*-6, *r*-5, and *r*-4 of the current column, and are ready for rounding at the end of each iteration.

3.4. Multiplication and Accumulation

The PMULLW instruction was used to accomplish the multiplications. This instruction performs a signed multiply of the four 16-bit components of a 64-bit value. With this instruction, only the low-order 16 bits of each of the four products are retained (the high-order 16 bits are discarded). This behavior is acceptable with the limitation that filter coefficient values must remain between 0 and 127; otherwise, significant bits of the product will be lost given the possible range of color values (0 to FFH).

To increase the possible range of the filter coefficients, you can use the PMADDWD instruction instead of the PMULLW instruction. PMADDWD multiplies the two signed 16-bit words contained in bits 0 through 31 of a given MMX register with memory or another MMX register. The 32-bit results of each of these two multiplications are added and placed in the low-order 32 bits of the destination register. A similar operation is performed on the upper 32 bits of the register/memory pair as well. The algorithm discussed above must be modified significantly to include the use of PMADDWD. Using the PMULLW instruction allows operating on four values simultaneously.

Additions (accumulations) are accomplished by using the PADDW instruction, which adds the four 16-bit components of the previously calculated products. The result of these multiply-accumulate operations must be rounded to the nearest whole value. This is accomplished by adding 0x80, (corresponding to 0.5 decimal), to the results of each of the four 16-bit results.

3.5. Packing the Data

After the multiplications and accumulations are complete, the final pixel value exists as four 16-bit fields within an MMX register, corresponding to alpha, red, green, and blue. These values must be packed down to 32-bit pixel values before being written back to the destination bitmap. In order to do this, the low-order eight bits of each sixteen bit result must be discarded, and the remaining upper eight bits be combined into a 32-bit value. This operation is accomplished through the use of the PSRLW and

PACKUSWB instructions. The PSRLW instruction performs a logical shift-right of the four 16-bit fields within an MMX register. By using a shift count of eight, the high-order eight bits of each 16-bit field are shifted to the low-order bit positions, leaving zeroes in the high-order bit positions.

Next, the PACKUSWB instruction combines the low-order bits from the four 16-bit fields of two MMX registers into two 32-bit values. In this manner, two pixel values are packed into one 64-bit MMX register, from two source MMX registers which each contain 32 bits of significant data. This final 64-bit value can then be written to the destination bitmap. This is illustrated in Figure 5. The final 64-bit value must be written as two 32-bit writes to memory, which involves a write, shift-right-by-32, and write. This is necessary because data is stored by the compiler in row-major order, where two successive pixels in a column are not stored physically in contiguous memory locations.

Figure 5. Packing Pixel Data

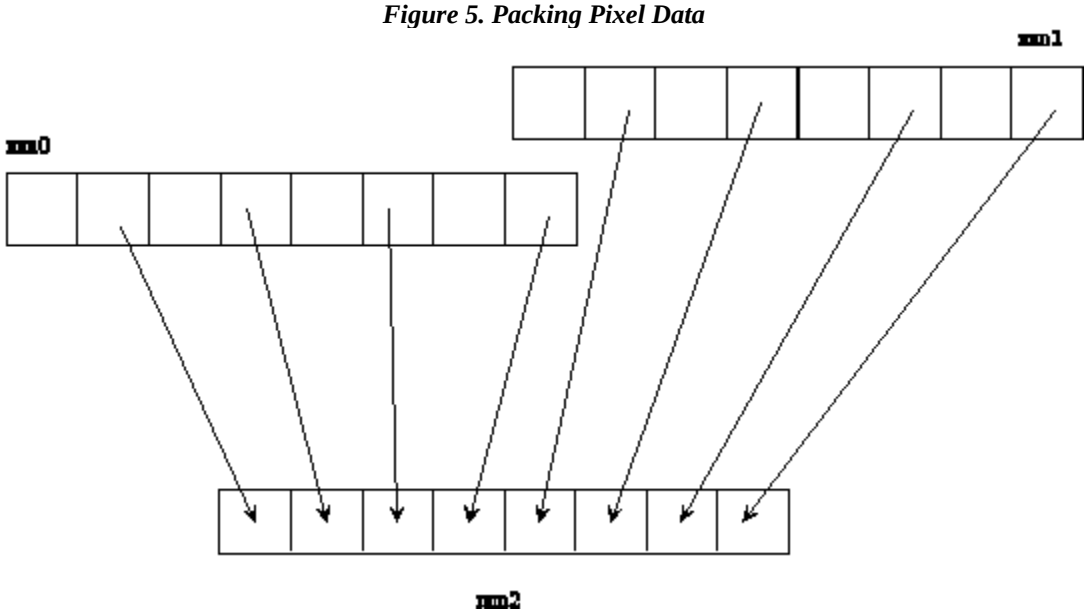


Figure 5 shows that the low-order 8 bits of the four 16-bit fields of MMX register MM1 are combined into the lower 32 bits of MM2. Likewise, the values contained in MM0 are combined into the upper 32 bits of MM2. In this manner, data is packed and made ready for the final write to the destination bitmap.

4.0. PERFORMANCE

This algorithm was tested on images with 72 columns and 58 rows, in which an average of 0.9 clocks per instruction was achieved in the critical routine. Once the caches became conditioned, the preconditioning code executed in 55 clock cycles, partially processing six pixels. The inner loop executed in 73 clock cycles. This corresponds to 24.3 clocks to completely process each 32-bit pixel. These values assume no misaligned accesses with a typical memory profile.

4.1. Optimization Techniques

The Pentium® processor contains a superscalar core, meaning it is possible to execute two instructions in a single clock cycle. The two execution pipelines of the Pentium processor are typically referred to as the U and V pipes. In general, the U pipe can execute any instruction, while only more basic instructions may be executed in the V pipe. Maintaining high instruction issue and execution rates obviously leads to improved performance. MMX technology is compliant with this superscalar nature and these pipelines can be made to execute in parallel. Taking advantage of this superscalar nature with proper instruction scheduling is the key to achieving desired performance goals.

One useful method for instruction scheduling is to define basic operations that must be performed and create a template which can be used to schedule instructions on a clock-by-clock basis to achieve optimal performance. For example, in this implementation seven basic operations are defined, as shown in Table 1.

Table 1. Basic Operations for the Column Filter

Operation	Definition
L	Load a 32-bit pixel value from the source bitmap.
U	Unpack a 32-bit pixel value by zero-extending each 8-bit component into 16-bit components.
M	Multiply values by the proper filter coefficients.
A	Perform the addition to accumulate products.
S	Store results to destination bitmap.
V	Move or copy the contents of one MMX register to another.
C	Clear the contents of an MMX register.

With the basic operations defined, a template can be created to aid in the instruction scheduling process. The template should be designed to show, on a clock-by-clock basis, all of the potential slots in which an instruction may execute. By following a few general guidelines, you can achieve very good results. The basic guidelines to keep in mind are:

1. Memory accesses can only be executed in the U pipe.
2. Minimize memory accesses, especially misaligned accesses.
3. All MMX instructions have a throughput of one instruction per clock and a latency of one instruction per clock, with the exception of the multiply instructions which have a latency of three clock cycles.
4. Use software pipelining to satisfy latency and pairing requirements.

5. Avoid register contention.
6. Utilize the code and data caches.

Table 2 shows an example of instruction scheduling for a basic set of operations assuming no loop unrolling is utilized.

Table 2. MMX Instructions to Process One Pixel

Clock Cycle		Pixel Operation	Clock Cycle		Pixel Operation
0	U	L0	5	U	
	V	C7		V	
1	U	U0	6	U	A43
	V			V	
2	U	V3	7	U	
	V			V	
3	U	M3	8	U	S4
	V			V	
4	U	L4T		U	
	V			V	

In Table 2, the 32-bit pixel value is loaded into register MM0 from the source bitmap and register MM7 is cleared in the U and V pipes of clock zero. In clock one, MM0 is unpacked against MM7 to yield a 64-bit value containing the four 8-bit values of the pixel (zero extended to sixteen bits). This value is copied into MM3 in clock two, and the multiplication of MM3 with a filter coefficient begins in clock three. A value is loaded from a temporary variable into MM4 in clock four. The multiplication begun in clock three completes in clock six, and that product is accumulated with the temporary variable (MM4) during clock six. Finally, the new result is stored to memory during clock eight. This example assumes all memory references hit in the cache.

Notice in the table that there are many "empty" slots in which useful work could be accomplished. For example, the V pipe in clock one is empty, as it is in clocks two, three, four, and so on. Through the use of this template, and experimentation, you can generally fill most if not all of the available slots, thus fully utilizing the available processing power.

Table 3 shows an example of instruction scheduling in which the loop is unrolled three times. This exposes more opportunities for parallel operations to be implemented, thus filling more slots.

Table 3. Template for the Column Filter Implementation

Clock Cycle		Pixel 0	Pixel 1	Pixel 2
0	U	L0		
	V	C7		
1	U		L1	
	V	U0		

2	U			L2
	V		U1	
3	U	V30		
	V			U2
4	U	M3		
	V			
5	U	L4T		
	V		V51	
6	U		M5	
	V		V60	
7	U		M6	
	V	A43		
8	U		L3T	
	V			V72
9	U	S4		
	V		A35	
10	U			M7
	V			V41
11	U			M4
	V		A63	
12	U			L5T
	V			V30
13	U			M3
	V			A57
14	U		S6	
	V			A45

Table 3 represents an example of the template used during the development of this algorithm. The table lists each clock cycle, the U and V pipe slots available within each clock, and the operation to be performed for pixel 0, 1, and 2. Consider first the column labeled Pixel 0. The first entry at clock 0 in the U pipe is L0. This represents a load from the source bitmap to MMX register MM0. This instruction must be issued in the U pipe because it performs a memory access. Along with this instruction, C7 is listed in the V pipe. This represents clearing register MM7 via the use of the PXOR instruction.

A zeroed register is required in the subsequent unpack operation. Unpacking could be performed against a memory location which had previously been initialized to zero, but this would have required that the unpack be performed in a U slot. With MM7 containing zero, the unpack operation (PUNPCKLBW) can be issued to the U or V pipes. Typically, U pipe slots are consumed before the V pipe slots are exhausted. The unpack operation occurs in the V slot of clock 1, and the unpacked pixel is copied to register MM3 as indicated in the U slot of clock 3.

The first multiplication occurs in the U slot of clock 4. Multiplications are performed in the U pipe because the filter coefficients are located in memory. This is because the filter coefficients are used often enough that you are assured they will remain in the data cache. MMX technology multiply instructions can be issued to the V pipe as well, as long as there are no associated memory accesses (both factors contained within MMX registers). The result of the multiplication will be placed in register MM3.

In the U slot of clock five, an intermediate result is loaded from a temporary variable located in memory for subsequent addition. This intermediate result is loaded into register MM4. Here again the data cache will contain the intermediate results due to the frequency of its use.

In the V slot of clock seven, the results of the multiplication stored in register MM3 are added with the intermediate value contained in register MM4, with the sum being stored in register MM4. At this particular point in the algorithm, the new intermediate value contained in register MM4 is written to memory for subsequent processing. Register MM4 is stored in the U slot of clock nine.

As shown in Table 3, there are several open slots in the Pixel 0 column wherein the processor can do additional work. For instance, the U slot of clock one is free in the Pixel 0 column. This is the utility of using the template. The template can help you quickly identify available processing slots that may be exploited. For this implementation, the available slots were used as shown in Table 3.

5.0. RESTRICTIONS

Presumably, you may choose any reasonable values for filter coefficients, or factors thereof. However, this implementation assumes that the product of any component value of a pixel and any filter coefficient will not exceed a 16-bit unsigned result, because only the low-order sixteen bits of the product are retained. Unfortunately, all of the MMX technology multiply instructions are signed, which further limits the product to fifteen bits. Since a color value may range from 0 to 255 (FFH), the maximum allowable value for a filter coefficient is 127 (decimal). Additionally, the sum of all of the products must not exceed a 16-bit signed value.

In general, filter coefficients can be arbitrarily sized (for arbitrary filter resolution) and signed. It is possible that after the multiply-accumulate steps are completed for a given component of a pixel, the result may be negative or may exceed some threshold value. For simplicity, no attempt was made to test for underflow or overflow of results. To investigate underflow or overflow testing, refer to the saturating add instructions available in the MMX instruction set.

In order to facilitate a simple code example, no attempt was made to support re-entrance in this implementation. In order to support re-entrance, you must eliminate the use of static variables within the algorithm. These are listed in Table 4.

Table 4. Static Variables Used in the Column Filter.

Variable	Definition
IMAGE_WIDTH	The width of the source bitmap (number of columns)
IMAGE_HEIGHT	The height of the source bitmap (number of rows)
FILTER_LENGTH	The number of filter coefficients
h0_6	Filter coefficients 0 and 6
h1_5	Filter coefficients 1 and 5
h2_4	Filter coefficients 2 and 4
h3	Filter coefficient 3
T0 through T8	Temporary variables 0 through 8

The image width, height, and filter length must be passed in as function parameters and referenced from the stack. A pointer to the array which holds the filter coefficients must also be used. The use of temporary variables must be completely eliminated, or accessed as stack variables. Edge conditions are not accounted for in this implementation.

APPENDIX A: SOURCE CODE

```
INCLUDE iammx.inc
    TITLE    MMX Code Column Filter
    .486P
include c:\msvc20\include\listing.inc
.model FLAT
;*************************************************************************/
;* Constant Definition(s)
IMAGE_WIDTH     EQU    72
IMAGE_HEIGHT    EQU    58
FILTER_LENGTH   EQU    7
;*************************************************************************/
;* Export Declaration(s)
PUBLIC  MMxColFilt
;*************************************************************************/
;* Data Segment Definition
_DATA    SEGMENT
; Filter coefficient table -
; the table is symmetric inasmuch as h0 == h6, h1 == h5, and h2 == h4.
; Assume filter coefficient table has been built by a previous process.
h0_6    SWORD    4 DUP (04)
h1_5    SWORD    4 DUP (24)
h2_4    SWORD    4 DUP (60)
h3      SWORD    4 DUP (80)
; Value of 0.1 (binary), added to the accumulated products of pixel values
; and coefficients, to round up to the next nearest bit.
rnd_off    SWORD    4 DUP (0080h)
; Temporary variables to store intermediate results for each iteration
; of the inner loop.
T0       QWORD         (0)
T1       QWORD         (0)
T2       QWORD         (0)
T3       QWORD         (0)
T4       QWORD         (0)
T5       QWORD         (0)
T6       QWORD         (0)
T7       QWORD         (0)
T8       QWORD         (0)
_DATA    ENDS
;*************************************************************************/
;* Text (code) Segment Definition
_TEXT SEGMENT
;-------------------------------------------------------------------------
; FUNCTION: MMxColFilt
;
;   Performs the column filtering on an input array, by multiplying each
;   pixel in a column by the appropriate filter coefficient, and accumulating
;   the results.
;
;   NOTE: In this implementation, function arguments for the pointer to the
;         array of filter coefficients, the array height and width, and the
;         number of filter coefficients are not used.  This function is not
;         re-entrant as implemented.
;
MMxColFilt PROC C PUBLIC USES eax ecx edx esi edi,
```

```
srcPtr    : DWORD,          ; pointer to source bitmap array
filtCoeff : DWORD,          ; pointer to array of filter coeff
dstPtr    : DWORD,          ; pointer to destination bitmap array
height    : WORD,           ; array height (number of rows)
wid       : WORD,           ; array width (number of columns)
filtLen   : WORD            ; filter array length
                            ; (number of filter coefficients)
    ;*************************************************************
    ; Perform the necessary setup including initializing pointers   *
    ; and loop counters.  The following register conventions are    *
    ; used.                                                         *
    ;                                                               *
    ; eax - offset into the source bitmap array                     *
    ; ecx - column loop counter                                     *
    ; edx - row loop counter                                        *
    ; esi - base address of the source bitmap array                 *
    ; edi - base address of the destination bitmap array            *
    ;                                                               *
    ; mm0 - pixel at [r,c]                                          *
    ; mm1 - pixel at [r+1,c]                                        *
    ; mm2 - pixel at [r+2,c]                                        *
    ;    (Other mmx registers are used for a variety of purposes.)  *
    ;*************************************************************
    ; initialize the column loop counter
    xor         ecx, ecx
    ; initialize source and destination pointers
    mov         esi, srcPtr
    mov         edi, dstPtr
    ;*************************************************************
    ; This begins the column loop.  Each iteration of this loop     *
    ; will cause the preconditioning code to execute, which         *
    ; initializes the temporary variables for the calculations      *
    ; performed in the inner loop.                                  *
    ;*************************************************************
col_loop:
precondition:
    ; initialize the row counter - preconditioning code handles
    ; rows 0 through 5
    mov         edx, 6
    ; calculate the offsets for indexing into the row of the source
    ; bitmap array
    lea         eax, DWORD PTR [ecx*4]
    ;*************************************************************
    ; In the comments that follow, the reader will notice the        *
    ; following conventions.                                        *
    ;                                                               *
    ;   Tx          - Temporary variable x.  There are nine         *
    ;                 temporary variables, T0 through T8.           *
    ;   BM[i+k,c]   - access to a particular pixel in the source    *
    ;                 bitmap array.  Rows are designated as i+k,    *
    ;                 and 'c' is the current column.  Each 32-bit   *
    ;                 pixel value is formatted as indicated below.  *
    ;                                                               *
    ;                31     24 23    16 15     8 7      0          *
    ;                +--------+--------+--------+--------+         *
    ;                | alpha  |   Red  | Green  |  Blue  |         *
    ;                +--------+--------+--------+--------+         *
```

```
;                                                                *
;   hx          - filter coefficient x.                          *
;                                                                *
; Unpacking a pixel converts it from a 32-bit value, to its      *
; four component 8-bit values zero-extended to 16-bits.  For     *
; example, a pixel unpacked from the 32-bit format shown above,  *
; will yield a 64-bit value as shown below.                      *
;                                                                *
;       63  56 55 48 47 40 39 32 31 24 23 16 15  8 7   0         *
;       +-----+-----+-----+-----+-----+-----+-----+-----+        *
;       |  0  |alpha|  0  |  R  |  0  |  G  |  0  |  B  |        *
;       +-----+-----+-----+-----+-----+-----+-----+-----+        *
;****************************************************************
;****************************************************************
; The first section of code reads in the first three pixels      *
; and performs all necessary calculations with these values.     *
;                                                                *
;    T0 = BM[0,c]*h0 + BM[1,c]*h1 + BM[2,c]*h2                   *
;    T1 = BM[1,c]*h0 + BM[2,c]*h1                                *
;    T2 = BM[2,c]*h0                                             *
;                                                                *
; In this implementation, a pixel is never read from the source *
; bitmap array more than once.                                   *
;****************************************************************
movd        mm2, [esi+eax+IMAGE_WIDTH*4*2]      ; load pixel from BM[2,c]
pxor        mm7, mm7                            ; clear mm7
movd        mm1, [esi+eax+IMAGE_WIDTH*4]        ; load pixel from BM[1,c]
punpcklbw   mm2, mm7                            ; unpack pixel BM[0,c]
movd        mm0, [esi+eax]                      ; load pixel from BM[0,c]
punpcklbw   mm1, mm7                            ; unpack pixel BM[1,c]
movq        mm3, mm2                            ; copy pixel BM[2,c] to mm3
punpcklbw   mm0, mm7                            ; unpack pixel BM[0,c]
pmullw      mm3, DWORD PTR h0_6                 ; mm3 <- BM[2,c]*h0
movq        mm4, mm2                            ; copy pixel BM[2,c] to mm4
pmullw      mm4, DWORD PTR h1_5                 ; mm4 <- BM[2,c]*h1
movq        mm5, mm1                            ; copy pixel BM[1,c] to mm5
pmullw      mm5, DWORD PTR h0_6                 ; mm5 <- BM[1,c]*h0
pmullw      mm2, DWORD PTR h2_4                 ; mm2 <- BM[2,c]*h2
pmullw      mm1, DWORD PTR h1_5                 ; mm1 <- BM[1,c]*h1
pmullw      mm0, DWORD PTR h0_6                 ; mm0 <- BM[0,c]*h0
paddw       mm5, mm4                            ; mm5 <- BM[2,c]*h1 + BM[1,c]*h0
movq        DWORD PTR T2, mm3                   ; T2 <- BM[2,c]*h0
;****************************************************************
; Read the next set of three pixels, and perform the necessary   *
; calculations.                                                  *
;                                                                *
;    T6 = BM[3,c]*h0 + BM[4,c]*h1 + BM[5,c]*h2                   *
;    T7 = BM[4,c]*h0 + BM[5,c]*h1                                *
;    T8 = BM[5,c]*h0                                             *
;****************************************************************
movd        mm6, [esi+eax+IMAGE_WIDTH*4*5] ; load pixel from BM[5,c]
paddw       mm1, mm2                       ; mm1 <- BM[2,c]*h2 + BM[1,c]*h1
movq        DWORD PTR T1, mm5              ; T1 <- BM[2,c]*h1 + BM[1,c]*h0
paddw       mm0, mm1                       ; mm0 <- BM[1,c]*h1 + BM[0,c]*h0
movd        mm1, [esi+eax+IMAGE_WIDTH*4*4] ; load pixel from BM[4,c]
punpcklbw   mm6, mm7                       ; unpack pixel BM[5,c]
```

```
    movq        DWORD PTR T0, mm0              ; T0 <- BM[2,c]*h2 + BM[1,c]*h1 +
BM[0,c]*h0
    movq         mm2, mm6                      ; save pixel BM[5,c]
    pmullw       mm6, DWORD PTR h0_6           ; mm6 <- BM[5,c]*h0
    punpcklbw    mm1, mm7                      ; unpack pixel BM[4,c]
    movd        mm0, [esi+eax+IMAGE_WIDTH*4*3  ; load pixel from BM[3,c]
    movq         mm4, mm2                      ; copy pixel BM[5,c]
    pmullw       mm4, DWORD PTR h1_5           ; mm4 <- BM[5,c]*h1
    movq         mm3, mm1                      ; copy pixel BM[4,c]
    pmullw       mm3, DWORD PTR h0_6           ; mm3 <- BM[4,c]*h0
    movq         mm5, mm2                      ; copy pixel BM[5,c]
    pmullw       mm5, DWORD PTR h2_4           ; mm5 <- BM[5,c]*h2
    punpcklbw    mm0, mm7                      ; unpack pixel BM[3,c]
    movq         DWORD PTR T8, mm6             ; T8 <- BM[5,c]*h0
    movq         mm7, mm1                      ; copy pixel BM[4,c]
    movq         mm6, mm0                      ; copy pixel BM[3,c]
    paddw        mm3, mm4                      ; mm3 <- BM[5,c]*h1 + BM[4,c]*h0
    pmullw       mm7, DWORD PTR h1_5           ; mm7 <- BM[4,c]*h1
    pmullw       mm6, DWORD PTR h0_6           ; mm6 <- BM[3,c]*h0
    ;*************************************************************
    ; Make additional calculations for previous three pixels.    *
    ;                                                            *
    ;    T3 = T0 + BM[3,c]*h3 + BM[4,c]*h4 + BM[5,c]*h5          *
    ;    T4 = T1 + BM[3,c]*h2 + BM[4,c]*h3 + BM[5,c]*h4          *
    ;    T5 = T2 + BM[3,c]*h1 + BM[4,c]*h2 + BM[5,c]*h3          *
    ;*************************************************************
    movq         DWORD PTR T7, mm3                ; T7 <- BM[5,c]*h1 + BM[4,c]*h0
    movq         mm4, mm2                         ; copy pixel BM[5,c]
    pmullw       mm4, DWORD PTR h1_5              ; mm4 <- BM[5,c]*h5
    paddw        mm7, mm5                         ; mm7 <- BM[5,c]*h2 + BM[4,c]*h1
    movq         mm3, mm1                         ; copy pixel BM[4,c]
    paddw        mm6, mm7                         ; mm6 <- BM[5,c]*h2 + BM[4,c]*h1
                                                  ; + BM[3,c]*h0
    pmullw       mm3, DWORD PTR h2_4              ; mm3 <- BM[4,c]*h4
    movq         mm7, mm0                         ; copy pixel BM[3,c]
    movq         mm5, DWORD PTR T0                ; mm5 <- BM[2,c]*h2 + BM[1,c]*h1
                                                  ; + BM[0,c]*h0
    pmullw       mm7, DWORD PTR h3                ; mm7 <- BM[3,c]*h3
    paddw        mm5, mm4                         ; mm5 <- BM[5,c]*h5 + BM[2,c]*h2
                                                  ; + BM[1,c]*h1 + BM[0,c]*h0
    movq         DWORD PTR T6, mm6                ; T6 <- BM[5,c]*h2 + BM[4,c]*h1
                                                  ; + BM[3,c]*h0
    paddw        mm3, mm5                         ; mm3 <- BM[5,c]*h5 + BM[4,c]*h4
                                                  ; + BM[2,c]*h2 + BM[1,c]*h1
                                                  ; + BM[0,c]*h0
    movq         mm5, DWORD PTR T1                ; mm5 <- BM[2,c]*h1 + BM[1,c]*h0
    movq         mm6, mm2                         ; copy pixel BM[5,c]
    pmullw       mm6, DWORD PTR h2_4              ; mm6 <- BM[5,c]*h4
    paddw        mm7, mm3                         ; mm7 <- BM[5,c]*h5 + BM[4,c]*h4
                                                  ; + BM[3,c]*h3 + BM[2,c]*h2
                                                  ; + BM[1,c]*h1 + BM[0,c]*h0
    pmullw       mm2, DWORD PTR h3                ; mm2 <- BM[5,c]*h3
    movq         mm4, mm1                         ; copy pixel BM[4,c]
    pmullw       mm4, DWORD PTR h3                ; mm4 <- BM[4,c]*h3
    movq         mm3, mm0                         ; copy pixel BM[3,c]
    pmullw       mm3, DWORD PTR h2_4              ; mm3 <- BM[3,c]*h2
    paddw        mm5, mm6                         ; mm5 <- BM[5,c]*h4 + BM[2,c]*h1
```

```
                                                    ; + BM[1,c]*h0
    pmullw      mm1, DWORD PTR h2_4                 ; mm1 <- BM[4,c]*h2
    pmullw      mm0, DWORD PTR h1_5                 ; mm0 <- BM[3,c]*h1
    paddw       mm4, mm5                            ; mm4 <- BM[5,c]*h4 + BM[4,c]*h3
                                                    ; + BM[2,c]*h1 + BM[1,c]*h0
    movq        mm5, DWORD PTR T2                   ; mm5 <- BM[2,c]*h0
    paddw       mm3, mm4                            ; mm3 <- BM[5,c]*h4 + BM[4,c]*h3
                                                    ; + BM[3,c]*h2 + BM[2,c]*h1
                                                    ; + BM[1,c]*h0
    movq        DWORD PTR T3, mm7                   ; T3 <- BM[5,c]*h5 + BM[4,c]*h4
                                                    ; + BM[3,c]*h3 + BM[2,c]*h2
                                                    ; + BM[1,c]*h1 + BM[0,c]*h0
    paddw       mm1, mm2                            ; mm1 <- BM[5,c]*h3 + BM[4,c]*h2
    movq        DWORD PTR T4, mm3                   ; T4 <- BM[5,c]*h4 + BM[4,c]*h3
                                                    ; + BM[3,c]*h2 + BM[2,c]*h1
                                                    ; + BM[1,c]*h0
    paddw       mm5, mm1                            ; mm5 <- BM[5,c]*h3 + BM[4,c]*h2
                                                    ; + BM[2,c]*h0
    paddw       mm0, mm5                            ; mm0 <- BM[5,c]*h3 + BM[4,c]*h2
                                                    ; + BM[3,c]*h1 + BM[2,c]*h0
    movq        DWORD PTR T5, mm0                   ; T5 <- BM[5,c]*h3 + BM[4,c]*h2
                                                    ; + BM[3,c]*h1 + BM[2,c]*h0
inner_loop:
    ;*************************************************************
    ; Calculate the offsets for indexing into the next           *
    ; row of the source and destination bitmap arrays.           *
    ;*************************************************************
    mov         eax, edx
    lea         eax, DWORD PTR [eax+eax*4]
    lea         eax, DWORD PTR [eax+eax*4]
    shl         eax, 4
    lea         eax, DWORD PTR [eax+ecx*4]
    ;*************************************************************
    ; Read the next set of three pixels, and perform the necessary  *
    ; calculations for pixels BM[r-6,c], BM[r-5,c], and BM[r-4,c].  *
    ;                                                               *
    ;    T0 = T3 + BM[r,c]*h6                                       *
    ;    T1 = T4 + BM[r,c]*h5 + BM[r+1,c]*h6                        *
    ;    T2 = T5 + BM[r,c]*h4 + BM[r+1,c]*h5 + BM[r+2,c]*h6         *
    ;*************************************************************
    movd        mm0, [esi+eax]                   ; load pixel from BM[r,c]
    pxor         mm7, mm7                        ; clear mm7
    movd        mm1, [esi+eax+IMAGE_WIDTH*4]    ; load pixel from BM[r+1,c]
    punpcklbw    mm0, mm7                       ; unpack pixel BM[r,c]
    movd        mm2, [esi+eax+IMAGE_WIDTH*4*2]  ; load pixel from BM[r+2,c]
    punpcklbw    mm1, mm7                       ; unpack pixel BM[r+1,c]
    movq         mm3, mm0                       ; copy pixel BM[r,c]
    punpcklbw    mm2, mm7                       ; unpack pixel BM[r+2,c]
    pmullw       mm3, DWORD PTR h0_6            ; mm3 <- BM[r,c]*h6
    movq         mm4, DWORD PTR T3              ; mm4 <- BM[r-1,c]*h5 + BM[r-2,c]*h4
                                                ; + BM[r-3,c]*h3 + BM[r-4,c]*h2
                                                ; + BM[r-5,c]*h1 + BM[r-6,c]*h0
    movq         mm5, mm1                       ; copy pixel BM[r+1,c]
    pmullw       mm5, DWORD PTR h0_6            ; mm5 <- BM[r+1,c]*h6
    movq         mm6, mm0                       ; copy pixel BM[r,c]
    pmullw       mm6, DWORD PTR h1_5            ; mm6 <- BM[r,c]*h5
    paddw        mm4, mm3                       ; mm4 <- BM[r,c]*h6 + BM[r-1,c]*h5
```

```
                                        ; + BM[r-2,c]*h4 + BM[r-3,c]*h3
                                        ; + BM[r-4,c]*h2 + BM[r-5,c]*h1
                                        ; + BM[0,c]*h0
movq        mm3, DWORD PTR T4           ; mm3 <- BM[r-1,c]*h4 + BM[r-2,c]*h3
                                        ; + BM[r-3,c]*h2 + BM[r-4,c]*h1
                                        ; + BM[r-5,c]*h0
movq        mm7, mm2                    ; copy pixel BM[r+2]
movq        DWORD PTR T0, mm4           ; T0 <- BM[r,c]*h6 + BM[r-1,c]*h5
                                        ; + BM[r-2,c]*h4 + BM[r-3,c]*h3
                                        ; + BM[r-4,c]*h2 + BM[r-5,c]*h1
                                        ; + BM[0,c]*h0
paddw       mm3, mm5                    ; mm3 <- BM[r+1,c]*h6 + BM[r-1,c]*h4
                                        ; + BM[r-2,c]*h3 + BM[r-3,c]*h2
                                        ; + BM[r-4,c]*h1 + BM[r-5,c]*h0
pmullw      mm7, DWORD PTR h0_6         ; mm7 <- BM[r+2,c]*h6
movq        mm4, mm1                    ; copy pixel BM[r+1,c]
pmullw      mm4, DWORD PTR h1_5         ; mm4 <- BM[r+1,c]*h5
paddw       mm6, mm3                    ; mm6 <- BM[r+1,c]*h6 + BM[r,c]*h5
                                        ; + BM[r-1,c]*h4 + BM[r-2,c]*h3
                                        ; + BM[r-3,c]*h2 + BM[r-4,c]*h1
                                        ; + BM[r-5,c]*h0
movq        mm5, DWORD PTR T            ; mm5 <- BM[r-1,c]*h3 + BM[r-2,c]*h2
                                        ; + BM[r-3,c]*h1 + BM[r-4,c]*h0
movq        mm3, mm0                    ; copy pixel BM[r,c]
pmullw      mm3, DWORD PTR h2_4         ; mm3 <- BM[r,c]*h4
paddw       mm5, mm7                    ; mm5 <- BM[r+2,c]*h6 + BM[r-1,c]*h3
                                        ; + BM[r-2,c]*h2 + BM[r-3,c]*h1
                                        ; + BM[r-4,c]*h0
movq        DWORD PTR T1, mm6           ; T1 <- BM[r+1,c]*h6 + BM[r,c]*h5
                                        ; + BM[r-1,c]*h4 + BM[r-2,c]*h3
                                        ; + BM[r-3,c]*h2 + BM[r-4,c]*h1
                                        ; + BM[r-5,c]*h0
paddw       mm4, mm5                    ; mm4 <- BM[r+2,c]*h6 + BM[r+1,c]*h5
                                        ; + BM[r-1,c]*h3 + BM[r-2,c]*h2
                                        ; + BM[r-3,c]*h1 + BM[r-4,c]*h0
;*************************************************************
; Perform the necessary calculations for pixels BM[r-3,c],    *
; BM[r-2,c], and BM[r-1,c].                                   *
;                                                             *
;    T3 = T6 + BM[r,c]*h3 + BM[r+1,c]*h4 + BM[r+2,c]*h5       *
;    T4 = T7 + BM[r,c]*h2 + BM[r+1,c]*h3 + BM[r+2,c]*h4       *
;    T5 = T8 + BM[r,c]*h1 + BM[r+1,c]*h2 + BM[r+2,c]*h3       *
;*************************************************************
movq        mm5, DWORD PTR T6           ; mm5 <- BM[r-1,c]*h2 + BM[r-2,c]*h1
                                        ; + BM[r-3,c]*h0
movq        mm6, mm2                    ; copy pixel BM[r+2,c]
pmullw      mm6, DWORD PTR h1_5         ; mm6 <- BM[r+2,c]*h5
paddw       mm3, mm4                    ; mm3 <- BM[r+2,c]*h6 + BM[r+1,c]*h5
                                        ; + BM[r,c]*h4 + BM[r-1,c]*h3
                                        ; + BM[r-2,c]*h2 + BM[r-3,c]*h1
                                        ; + BM[r-4,c]*h0
movq        mm4, mm1                    ; copy pixel BM[r+1,c]
pmullw      mm4, DWORD PTR h2_4         ; mm4 <- BM[r+1,c]*h4
movq        mm7, mm0                    ; copy pixel BM[r,c]
pmullw      mm7, DWORD PTR h3            ; mm7 <- BM[r,c]*h3
paddw       mm5, mm6                    ; mm5 <- BM[r+2,c]*h5 + BM[r-1,c]*h2
                                        ; + BM[r-2,c]*h1 + BM[r-3,c]*h0
```

```
    movq        DWORD PTR T2, mm3           ; T2 <- BM[r+2,c]*h6 + BM[r+1,c]*h5
                                            ; + BM[r,c]*h4 + BM[r-1,c]*h3
                                            ; + BM[r-2,c]*h2 + BM[r-3,c]*h1
                                            ; + BM[r-4,c]*h0
    paddw       mm4, mm5                    ; mm4 <- BM[r+2,c]*h5 + BM[r+1,c]*h4
                                            ; + BM[r-1,c]*h2 + BM[r-2,c]*h1
                                            ; + BM[r-3,c]*h0
    movq        mm3, mm2                    ; copy pixel BM[r+2,c]
    pmullw      mm3, DWORD PTR h2_4         ; mm3 <- BM[r+2,c]*h4
    paddw       mm7, mm4                    ; mm7 <- BM[r+2,c]*h5 + BM[r+1,c]*h4
                                            ; + BM[r,c]*h3 + BM[r-1,c]*h2
                                            ; + BM[r-2,c]*h1 + BM[r-3,c]*h0
    movq        mm6, DWORD PTR T7           ; mm6 <- BM[r-1,c]*h1 + BM[r-2,c]*h0
    movq        mm5, mm1                    ; copy pixel BM[r+1,c]
    pmullw      mm5, DWORD PTR h3           ; mm5 <- BM[r+1,c]*h3
    movq        mm4, mm0                    ; copy pixel BM[r,c]
    pmullw      mm4, DWORD PTR h2_4         ; mm4 <- BM[r,c]*h2
    paddw       mm6, mm3                    ; mm6 <- BM[r+2,c]*h4 + BM[r-1,c]*h1
                                            ; + BM[r-2,c]*h0
    movq        DWORD PTR T3, mm            ; T3 <- BM[r+2,c]*h5 + BM[r+1,c]*h4
                                            ; + BM[r,c]*h3 + BM[r-1,c]*h2
                                            ; + BM[r-2,c]*h1 + BM[r-3,c]*h0
    movq        mm3, mm2                    ; copy pixel BM[r+2,c]
    pmullw      mm3, DWORD PTR h3           ; mm3 <- BM[r+2,c]*h3
    paddw       mm5, mm6                    ; mm5 <- BM[r+2,c]*h4 + BM[r+1,c]*h3
                                            ; + BM[r-1,c]*h1 + BM[r-2,c]*h0
    paddw       mm4, mm5                    ; mm4 <- BM[r+2,c]*h4 + BM[r+1,c]*h3
                                            ; + BM[r,c]*h2 + BM[r-1,c]*h1
                                            ; + BM[r-2,c]*h0
    movq        mm5, DWORD PTR T8           ; mm5 <- BM[r-1,c]*h0
    movq        mm6, mm1                    ; copy pixel BM[r+1,c]
    pmullw      mm6, DWORD PTR h2_4         ; mm6 <- BM[r+1,c]*h2
    movq        mm7, mm0                    ; copy pixel BM[r,c]
    pmullw      mm7, DWORD PTR h1_5         ; mm7 <- BM[r,c]*h1
    paddw       mm5, mm3                    ; mm5 <- BM[r+2,c]*h3 + BM[r-1,c]*h0
    ;***************************************************************
    ; Perform the necessary calculations for pixels BM[r,c],       *
    ; BM[r+1,c], and BM[r+2,c].                                    *
    ;                                                              *
    ;    T6 = BM[r  ,c]*h0 + BM[r+1,c]*h1 + BM[r+2,c]*h2           *
    ;    T7 = BM[r+1,c]*h0 + BM[r+2,c]*h2                          *
    ;    T8 = BM[r+2,c]*h0                                         *
    ;***************************************************************
    movq        DWORD PTR T4, mm4           ; T4 <- BM[r+2,c]*h4 + BM[r+1,c]*h3
                                            ; + BM[r,c]*h2 + BM[r-1,c]*h1
                                            ; + BM[r-2,c]*h0
    movq        mm3, mm2                    ; copy pixel BM[r+2,c]
    pmullw      mm3, DWORD PTR h2_4         ; mm3 <- BM[r+2,c]*h2
    movq        mm4, mm1                    ; copy pixel BM[r+1,c]
    pmullw      mm4, DWORD PTR h1_5         ; mm4 <- BM[r+1,c]*h1
    paddw       mm6, mm5                    ; mm6 <- BM[r+2,c]*h3 + BM[r+1,c]*h2
                                            ; + BM[r-1,c]*h0
    pmullw      mm0, DWORD PTR h0_6         ; mm0 <- BM[r,c]*h0
    paddw       mm7, mm6                    ; mm7 <- BM[r+2,c]*h3 + BM[r+1,c]*h2
                                            ; + BM[r,c]*h1 + BM[r-1,c]*h0
    movq        mm5, mm2                    ; copy pixel BM[r+2,c]
    pmullw      mm5, DWORD PTR h1_5         ; mm5 <- BM[r+2,c]*h1
```

```
    pmullw      mm1, DWORD PTR h0_6          ; mm1 <- BM[r+1,c]*h0
    paddw       mm4, mm3                     ; mm4 <- BM[r+2,c]*h2 + BM[r+1,c]*h1
    movq        DWORD PTR T5, mm7            ; T5 <- BM[r+2,c]*h3 + BM[r+1,c]*h2
                                             ; + BM[r,c]*h1 + BM[r-1,c]*h0
    paddw       mm0, mm4                     ; mm0 <- BM[r+2,c]*h2 + BM[r+1,c]*h1
                                             ; + BM[r,c]*h0
    pmullw      mm2, DWORD PTR h0_6          ; mm2 <- BM[r+2,c]*h0
    movq        DWORD PTR T6, mm0            ; T6 <- BM[r+2,c]*h2 + BM[r+1,c]*h1
                                             ; + BM[r,c]*h0
    paddw       mm1, mm5                     ; mm1 <- BM[r+2,c]*h1 + BM[r+1,c]*h0
    movq        DWORD PTR T7, mm1            ; T7 <- BM[r+2,c]*h1 + BM[r+1,c]*h0
    movq        DWORD PTR T8, mm2            ; T8 <- BM[r+2,c]*h0
    ;*************************************************************
    ; Each componenet 16-bit value of the pixel (aRGB) must be    *
    ; rounded up to the next nearest bit.  This is accomplished by*
    ; adding 80H to each 8-bit value.  This effectively adds one  *
    ; half to each value (0.1 binary), since the binary point is  *
    ; between bit 7 and bit 8.  After rounding, the high-order    *
    ; eight bits of the 16-bit result are taken; the low-order    *
    ; eight bits are discarded.                                   *
    ;                                                             *
    ;       63  56 55 48 47 40 39 32 31 24 23 16 15  8 7   0      *
    ;       +-----+-----+-----+-----+-----+-----+-----+-----+      *
    ;       |  ah |  al |  Rh |  Rl |  Gh |  Gl |  Bh |  Bl |      *
    ;       +-----+-----+-----+-----+-----+-----+-----+-----+      *
    ;                                                             *
    ;       |     +     |     +     |     +     |     +     |      *
    ;                                                             *
    ;       63  56 55 48 47 40 39 32 31 24 23 16 15  8 7   0      *
    ;       +-----+-----+-----+-----+-----+-----+-----+-----+      *
    ;       |  0  | 0x80|  0  | 0x80|  0  | 0x80|  0  | 0x80|      *
    ;       +-----+-----+-----+-----+-----+-----+-----+-----+      *
    ;                                                             *
    ;       |           |           |           |           |      *
    ;       +-----+-----+-----+-----+-----+-----+-----+-----+      *
    ;             |           |           |           |            *
    ;             |           |           |           +--+         *
    ;             |           |           +---------+    |         *
    ;             |           +---------------+     |    |         *
    ;             +--------------------+    |     |    |         *
    ;                                  |    |     |    |         *
    ;                                  V    V     V    V         *
    ;       63  56 55 48 47 40 39 32 31 24 23 16 15  8 7   0      *
    ;       +-----+-----+-----+-----+-----+-----+-----+-----+      *
    ;       |   x |  x  |  x  |  x  |  a  |  R  |  G  |  B  |      *
    ;       +-----+-----+-----+-----+-----+-----+-----+-----+      *
    ;                                                             *
    ;*************************************************************
    ;*************************************************************
    ; The pixel values stored in T0, T1, and T2 are completely    *
    ; calculated (except for round-off).  Add round-off and store *
    ; to destination bitmap array.                                *
    ;*************************************************************
    movq        mm4, DWORD PTR rnd_off        ; get the round-off value
    movq        mm0, DWORD PTR T0             ; mm0 <- BM[r-6,c] (64-bits)
    movq        mm1, DWORD PTR T1             ; mm1 <- BM[r-5,c] (64-bits)
    paddw       mm0, mm4                      ; mm0 <- mm0 + round-off
```

```
    movq        mm2, DWORD PTR T2               ; mm2 <- BM[r-4,c] (64-bits)
    paddw       mm1, mm4                        ; mm1 <- mm1 + round-off
    paddw       mm2, mm4                         ; mm2 <- mm2 + round-off
    ;*************************************************
    ;* Update
    ;*************************************************
    psrlw       mm0, 8                          ; rotate each 16-bit component
                                                ; of 64-bit result right by
                                                ; eight bits (discard low-
                                                ; order eight bits)
    psrlw       mm1, 8                          ; rotate each word component
    psrlw       mm2, 8                          ; rotate each word component
    pxor        mm7, mm7                        ; clear mm7
    packuswb    mm0, mm1                        ; pack high-order byte of each
                                                ; word component into 32-bit
                                                ; result
    add         edx, 3                          ; increment the row counter
    movd       [edi+eax-IMAGE_WIDTH*4*6], mm0   ; store BM[r-6,c] (final result)
    packuswb    mm2, mm7                        ; pack high-order bytes
    psrlq       mm0, 32                         ; discard low-order 32-bit word
                                                ; (high-order word contains pixel
                                                ; value)
    movd       [edi+eax-IMAGE_WIDTH*4*5], mm0   ; store BM[r-5,c] (final result)
    movd       [edi+eax-IMAGE_WIDTH*4*4], mm2   ; store BM[r-4,c] (final result)
    ; finished with rows?
    cmp         edx, IMAGE_HEIGHT - 1
    jl          inner_loop
    ; finished with columns?
    inc         ecx
    cmp         ecx, IMAGE_WIDTH
    jl          col_loop
    ; release MMx context
    emms
    ; return to caller
    ret    0
MMxColFilt ENDP
_TEXT    ENDS
END
```